Solution for Insecure data transmission of IoT devices over MQTT protocol.

Ravindu M.A.C IT20090326  
 *Sri Lanka Institue of Information Technology*New Kandy Road, Malabe, Sri Lanka  
it20090326@my.sllit.lk

***Abstract—MQTT (Message Queuing Telemetry Transport) is a widely used data transmission protocol among IoT devices. It uses publisher-subscriber method to transmit messages from a device to device. Once a device transmits a message to a relevant topic, all the devices that are subscribed to that topic receives the message. The data that are transmitted via MQTT takes the format of plain text by default. Even though to ensure security CA certificates can be used, it has few practicality issues when it comes to low-powered IoT devices. MQTT is widely used my DIY electronic enthusiasts to build their own DIY home automation systems. Most of them are not aware of insecure data transmission vulnerability presence in MQTT protocol. As a solution for this I am proposing a new library called MQTTS (Message Queuing Telemetry Transport Secure). MQTTS is a lightweight and flexible library which runs on top of MQTT library ensuring the end-to-end security between IoT devices. When using MQTT, any device or any person on the network, can send or receive messages easily by publishing or subscribing to a certain topic. This could lead to a severe security vulnerability in certain scenarios. MQTTS is capable of mitigating this vulnerability by sharing an asymmetric cryptographic key between the client and the server and using rolling code to prevent replay attacks.***

***Keywords— MQTT, Encryption, RSA, Python3***

# Introduction

MQTT is a light-weight protocol which is used by DIY enthusiasts to build IoT systems [1]. MQTT uses publisher-subscriber method to pass data/message from a device to device. MQTT service can be running on a central hub or on a cloud platform [2]. To run the MQTT service the device should poses a decent amount of power. DIY enthusiasts use single board computers such as Raspberry pi, Pine 64, Orange pi, Banana pi etc [3]. After implementing the MQTT service, any IoT device on the same network can use the MQTT service to pass data/messages. To pass data/messages via MQTT, clients use topic to send data/messages and the to receive those data clients must subscribe to the topic [4]. When a client sends a message via a topic, the message first arrives to the device that runs MQTT service. The MQTT service then emits the message to the topic and all the clients that are subscribed to the topic will receive the message. A client can send messages to multiple topics and receivers are able to subscribe to multiple topics to receive messages. The maximum amount of data that can be transferred via MQTT is 260MB [5]. MQTT is rather easy to implement when compared to socket and port architecture [6]. The data that gets transferred via MQTT is not secured by default. Due to this, anyone in the same network is able to capture MQTT messages using a packet capture software like Wire Shark.

# Methodology

MQTT uses either local area network or internet to send and receive data. If the MQTT server or the service provider has not configured a CA certificate, all the data that are transmitted via MQTT is plain text. A packet capturing tool like Wire Shark is able to show the topic and the transmitted message.

A screenshot of a computer

Description automatically generated

Figure 1

According to the above image (figure 1) it is clear that the unencrypted data is transmitted by the MQTT clients. This is a severe vulnerability. Due to this vulnerability attackers are able to carry out replay attacks. If the system is an IoT access control system, due to this vulnerability attackers are able to gain access after performing a replay attack. This is a critical scenario.

As a solution for this, CA certificate can be used. Before deploying the IoT device and the server, programmers must configure the CA certificate on both clients and the server. Using only a CA certificate does not provide a hundred percent security.

Configuring a CA certificate may not be easy to amateur IoT programmers. Especially if the programmer is not aware about the security concepts. Configuring the CA certificate on multiple devices takes time and if the CA certificate configuration is unsuccessful, the device security will be not ensured. Even though a CA certificate is used, the device is still vulnerable to replay attacks [7].

Due to improper validation methodologies, any client is able to pass data to another client or listen to any topic resulting rogue data generation. This could lead to a serious security issue.

# SOLUTION

Wire Shark is able to how all of the traffic that is generated by MQTT clients and the server. If the data is not subjected to encryption, all of the data that gets transmitted will be transmit in plain text. As a result of this the data that gets transmitted over the network is visible to all of the hosts. An attacker is able to capture those data packets by using a packet capturing software such as Wire Shark and the attacker is able to see the topic and the message that is transmitted [8].

After gathering the necessary information such as the topic and the message, an attacker is able to carry out a replay attack. The MQTT server or the clients are not able to separate traffic from legit clients or rogue clients.

Data encryption is one of the major concepts when it comes to data security. Data encryption can be applied to MQTT manually up to a particular extend. Asymmetric encryption is more secure that the symmetric encryption methods as it uses a public key to encrypt and a private key to decrypt [9]. Even though the data get transmitted is subjected to encryption, it is still vulnerable to replay attacks since the keys does not change and when the same key encrypts the same data it generates a constant encrypted message.

# DISSCUSSION

Ensuring the maximum MQTT data protection is mandatory when it comes to critical IoT devices. As a solution for MQTT security MQTTS can be implemented. MQTT is a light-weight protocol that runs on top of MQTT written in python3.

MQTTS consists with 5 stages namely, Key generation, key transmission, rolling code generation, rolling code transmission, rolling code acceptance.

MQTTS uses both asymmetric encryption and rolling key mechanism to prevent replay attacks. Rolling code/code hopping is a mechanism that uses in vehicle smart keys to prevent replay attacks [10]. It uses multiple codes to lock and unlock the vehicle. If the vehicle gets unlocked successfully, then the key fob hops to the next code when unlocking the vehicle. Due to rolling code mechanism, attackers are unable to use the recorded unlocking signal sequence to unlock the vehicle again. MQTTS uses the rolling code mechanism to prevent replay attacks by appending a numeric value to the message. Every time the server receives a message from a client, it increments it’s value by a pre-defined value. As a result of this, attackers are unable to perform replay attacks.

# MQTTS dependencies

Several python3 modules are required to run MQTTS. The setup of the environment before launching MQTTS is necessary. The client configuration and the server configuration are as follows.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

When installing python modules on Linux, it is necessary to use pip3 instead of pip.

## Creating a MQTTS client

MQTTS client can be made by using the MQTTS\_CLIENT class. With the help of MQTTS\_CLIENT class, single script is capable of creating multiple MQTTS clients as they all run on an isolated thread. The MQTTS\_CLIENT constructor consists with several paraments. The image below (figure 2) shows an example of an MQTTS client.

A screenshot of a computer

Description automatically generated

Figure 1

The parameters of MQTTS\_CLIENT are,

str:client\_id, str:client\_name, str:server\_ip, int:server\_port

The image below (figure 3) shows the code snippet of the MQTTS\_CLIENT constructor.

A screenshot of a computer

Description automatically generated

Figure 2

## Creating MQTTS\_SERVER

MQTTS server can be created by using the MQTTS\_SERVER class. MQTTS\_SERVER constructor consists of several parameters. The image below (figure 4) shows and example of MQTTS server. To avoid unexpected server crashes, the code snippet is enclosed with a try except block.

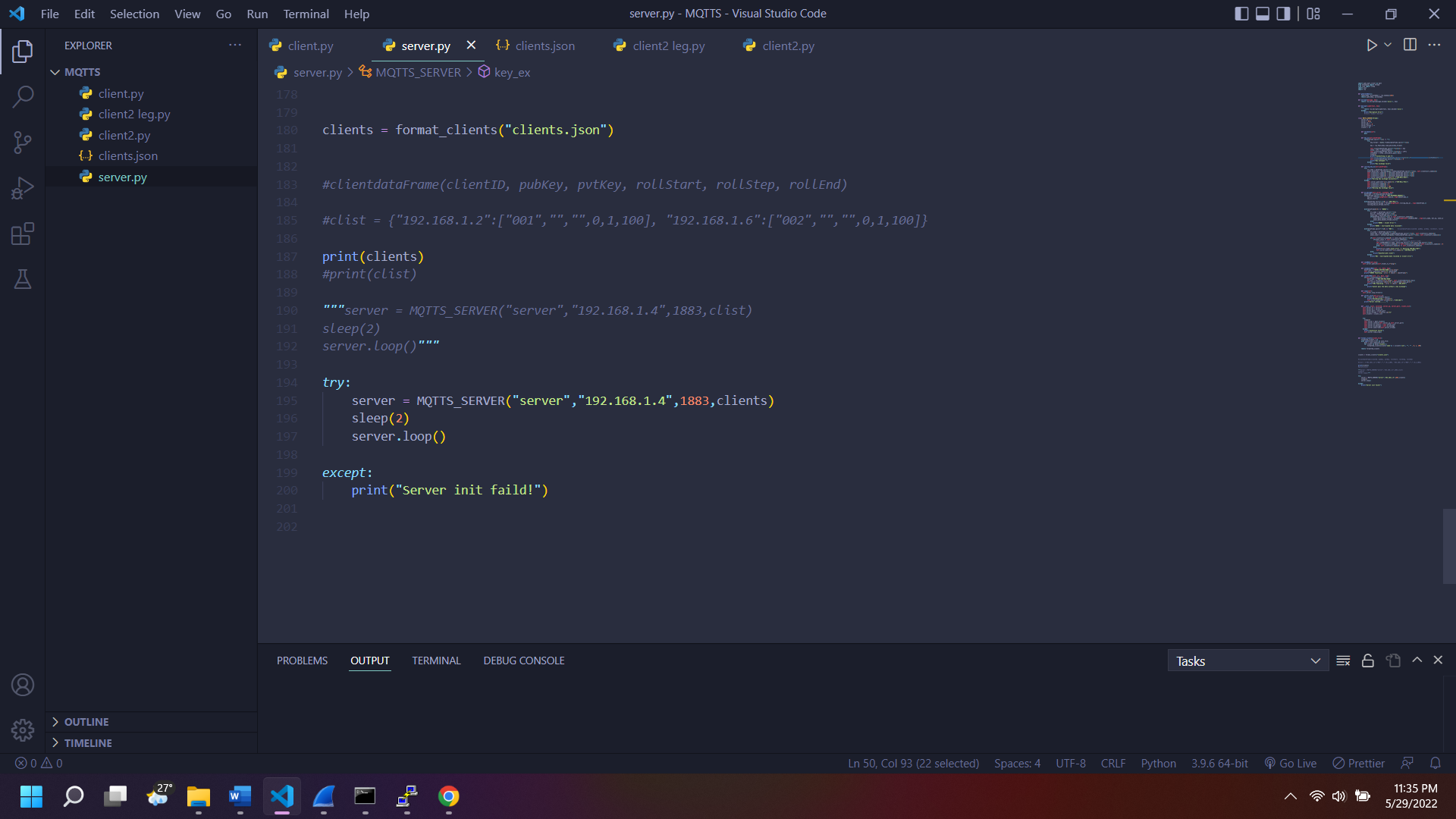


Figure 3

The parameters of MQTTS\_Server are,

str:server\_name, str:server\_ip, int:server\_port, dictionary:client\_list

Client list is a pre-defined client list architecture by MQTTS. To make a valid client list, format\_clients() method can be used. It takes a parameter of a JSON file. The following code snippet (figure 5) shows the client\_format() method.

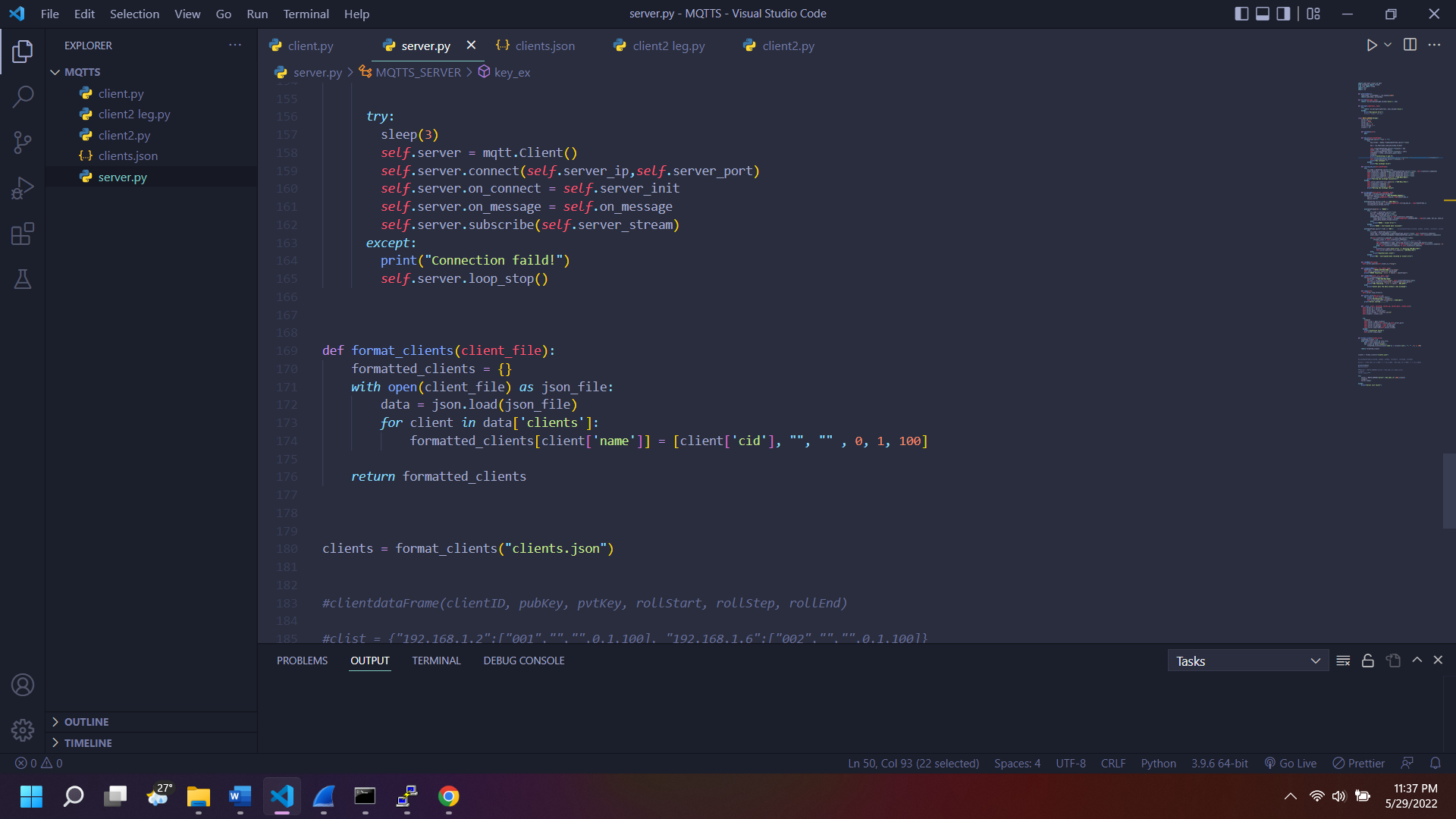


Figure 4

The structure of a valid client is as follows

{str:client\_name : [str:client\_id, obj:public\_key, obj:privte\_key, int:rolling\_start, int:rolling\_increment, int:rolling\_end]}

The image below (figure 6) shows an example of client JSON file.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 5

## MQTTS message flags

MQTTS uses several different flags in the beginning of the message to identify the status of the message by the MQTTS server and the clients.

SVR-INIT: send to all client by the server that the server has initialized.

KEY\_EXCHANGE\_REQUEST: send by the client to the server to initiate the key exchange sequence with public key attached.

SVR-KEY-EXCHANGE\_REPLY: send by the server to the client with the public key

INIT-ROLL: send by the client to the server to start the rolling code initiation sequence attaching the start value, increment value, ending value of the rolling code.

SVR-ROLL-SUCC: Send by the server to the client to state that the rolling key is accepted by the server.

SVR-ROLL-FAIL: Send by the server stating that the rolling code initiation failed by the server.

UNENC: send by both client and the server. UNENC represents an unencrypted message.

ENC: send by both client and the server. ENC represents encrypted message.

## Key generation and transmission stage

MQTTS uses asymmetric RSA algorithm to generate public and private keys [11]. Key generation can be invoked by using key\_ex() method by the client program. Client is the first to initiate the key exchange sequence. Key exchange takes place in a different thread. Therefore, key exchange does not affect the message send and receive loop. After generating a private and a public key pair, client will save the private key and, on its memory, and the public key will be transmitted to the server with “KEY\_EXCHNAGE\_REQUEST” flag. After receiving the key exchange request by a client, the server will start the key exchange sequence by generating a dedicated private and a public key pair to a particular client. The key exchange sequence is handled by the server in a separate thread to make sure that the server is open to send or receive messages from other clients and handle message forwarding without an interruption. After generating a key pair, the server will save the private key to a particular client and the transmit the public key to the client with the flag of “SVR-KEY\_EXCHANGE\_REPLY”. After receiving the reply form the server, the client will save the private key. If the key exchange fails, the client will re-initiate the key exchange sequence once again. The image below shows the code snippet of MQTTS key exchange of client (figure 7) and the server (figure 8).

Text

Description automatically generated

Figure 6

A screenshot of a computer

Description automatically generated

Figure 7

## Rolling code generation and rolling code transmission

Rolling code is the replay attack mechanism in MQTTS. Rolling code initiation takes place after a successful key exchange with the MQTTS server. Without a proper key exchange, rolling code initiation and exchange will not be take place.

Rolling code will be set to 0 by default and it will get incremented by 1 for every successful message transmission. Rolling code initiation can be override manually by using init\_rolling\_code(code start value, increment value, code end value) method. Code start is the initial value of the code. Increment value is the number that increments the code for every successful message. Code end value is the ending value of the rolling codes. After reaching the end value, the client will start over from the initial value that is configured. After invoking the init\_rolling\_code() method, the client will transmit the key start value, increment value and the ending value to the server as an encrypted message using the shared public key and will attach the “SVR-INT-ROLL” flag. After receiving the message, the server will set the new rolling code start, increment and ending values according to the relevant client. Rolling code transmission and acceptance by client and the server takes place in a different thread to isolate the MQTTS main sequence. At the end of the sequence the server will emit “SVR-ROLL-SUCC” flag to indicate that the rolling code initiation success. If an error occurred during this process, the server would emit “ROLL-FAIL” and the client will re-initiate the process. If the code that sent by the client appending the rolling code value to the message is different from the expecting value by the server, the server will emit “SVR-ROLL-RST” to reset the rolling code value to start over from the initial value. Images below shows the code snippet for rolling key handling by the client (figure 9) and the server (figure 10).

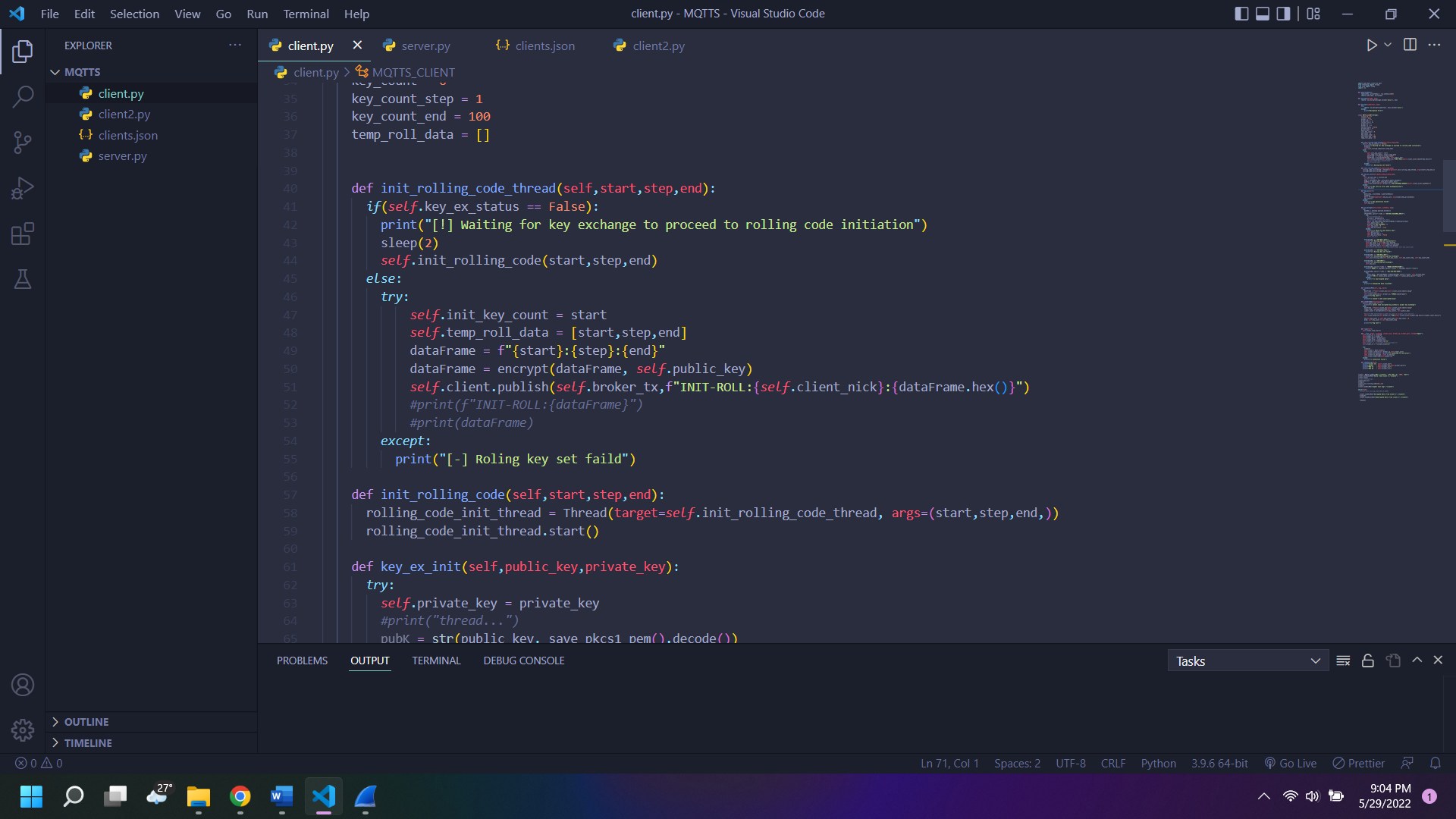


Figure 8

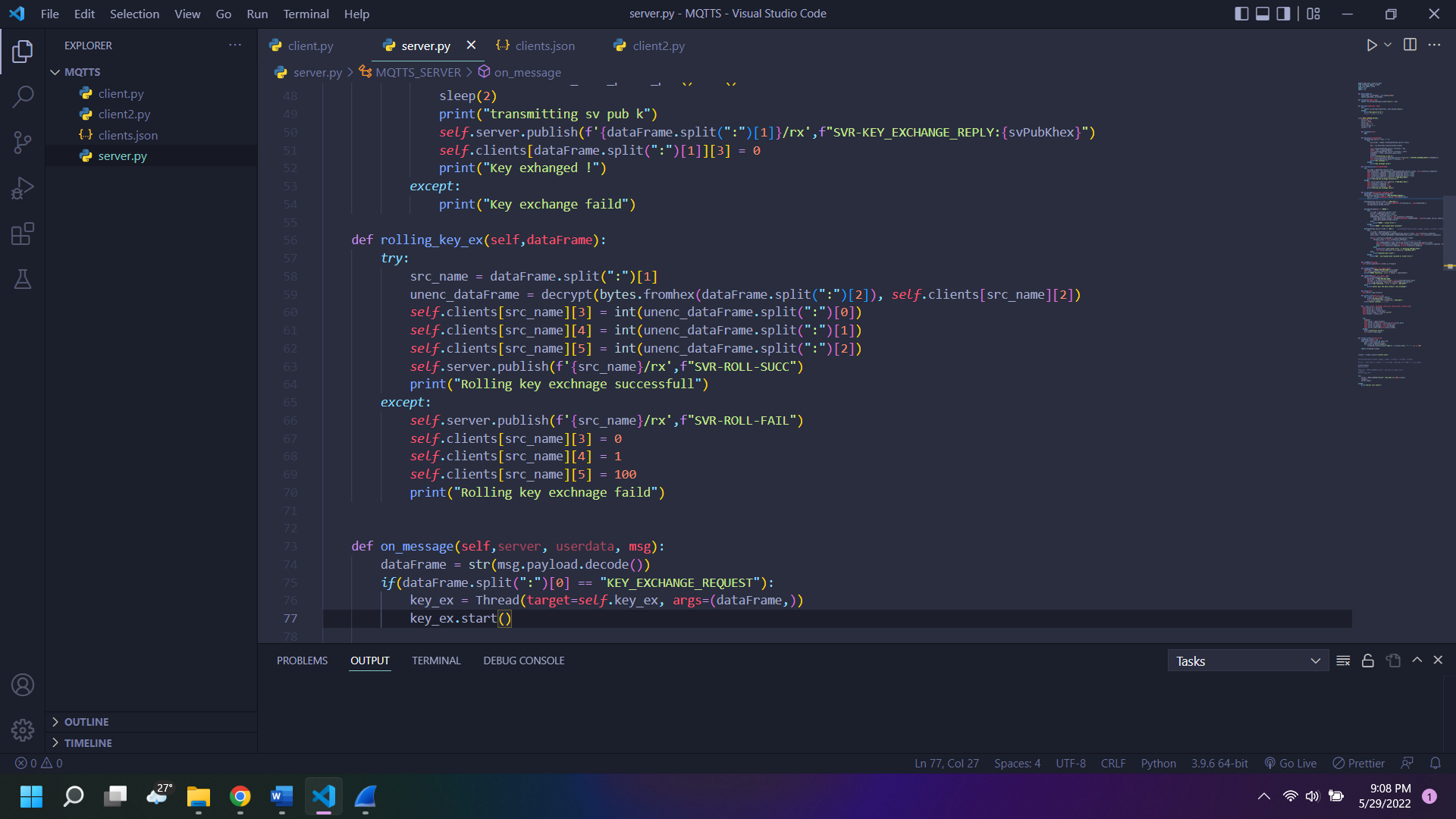


Figure 9

## Sending unencrypted messages

MQTTS is still capable of transmitting unencrypted messages in case of a key exchange failure. Unencrypted message transmission is not depreciated due to low end IoT devices and legacy systems are not capable of using MQTTS due to hardware or software incompatibility. The format of an unencrypted message is shown below.

UNENC:client\_id:client\_name:receiver\_name:message

Unencrypted message is sent with the UNENC flag. The server will forward the unencrypted message to the particular client once it received from a client. The image below (figure 11) shows the code snippet for sending an unencrypted message from a client. Unencrypted message can be set by invoking sendUnencMSG(message, destinantion) method.

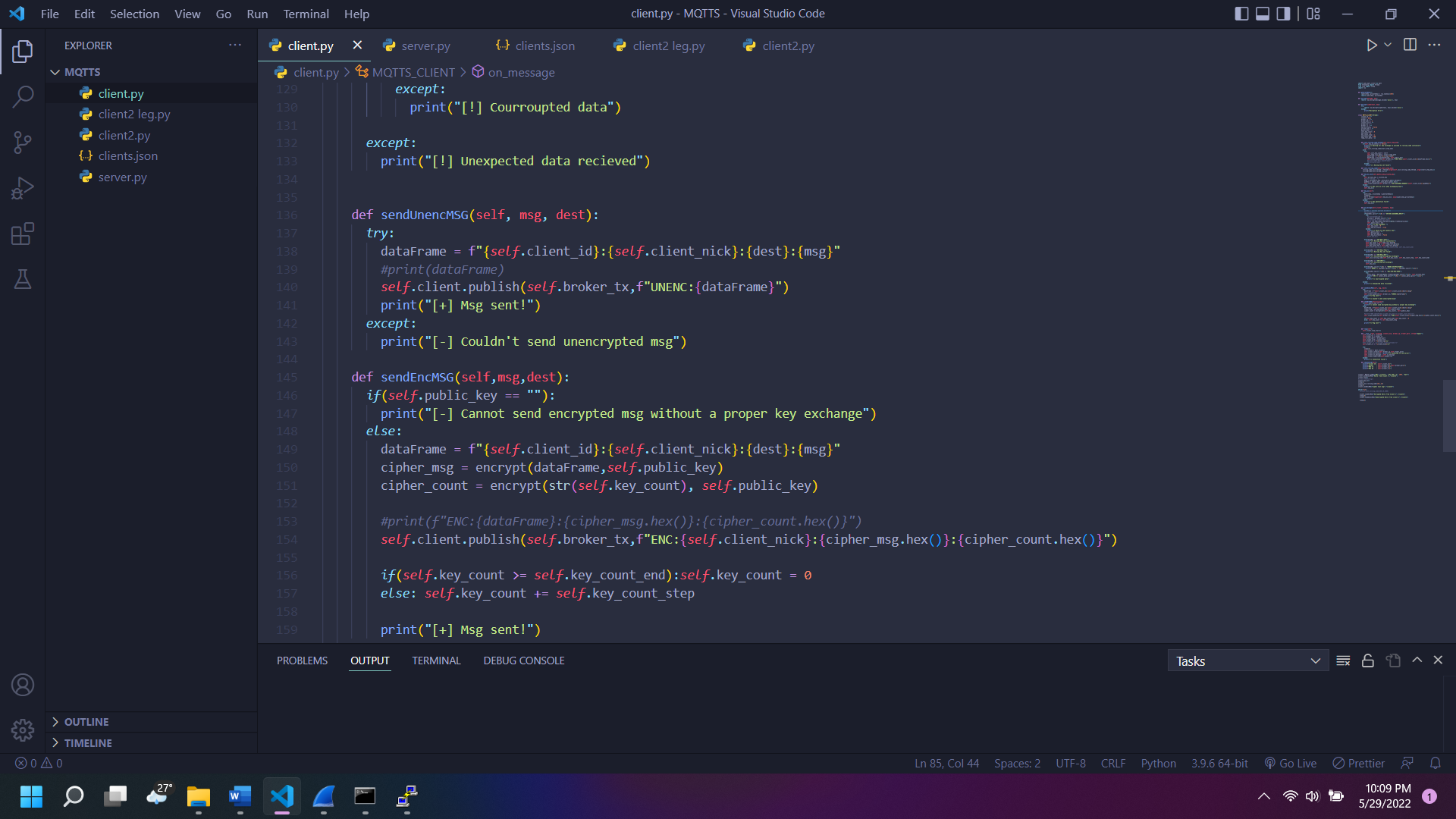


Figure 10

## Sending encrypted messages

Encrypted message sending is the priority of MQTTS. Before sending an encrypted message, the client and the server must make sure that the keys and the rolling code is successfully exchanged. If the key exchange is unsuccessful, MQTTS will emit and error stating that the encrypted message can not be sent without exchanging keys. The format of an encrypted message is shown below.

ENC:client\_name:encrypted\_message

Encrypted messages is passed with the ENC flag. The encrypted message contains the rolling code which is being used. Encrypted messages can be sent by invoking sendEncMSG(message, destination) method. The following code snippet (figure 12) shows the handing of an encrypted message by a client.

Text

Description automatically generated

Figure 11

# Future developments

MQTTS is currently can be implemented on IoT devices that supports desktop python3. Every single board computer on the market is capable of implementing MQTTS. Low end IoT devices often use Micro-Python instead of regular Python as the programming language. Micro-Python is currently lacking dependencies such as python RSA module, Paho-MQTT client and Micro-Python is platform dependent [12]. As a result of this some IoT development boards such as ESP8266 is not capable of performing multi-threading. Due to lack of modules, MQTTS currently unavailable for low end IoT platforms. Future of MQTTS is introducing the MQTTS\_LITE to make sure all the low end IoT platforms run MQTTS flawlessly.

# REFERENSES

|  |  |
| --- | --- |
| [1] | A. Agarwal, "DIY Control any home appliance with the ESP8266 | Reyax MQTT," hackster.io, 11 7 2021. [Online]. Available: https://www.hackster.io/akarsh98/diy-control-any-home-appliance-with-the-esp8266-reyax-mqtt-64bda0. [Accessed 30 5 2022]. |
| [2] | MQTT, "What is MQTT," [Online]. Available: https://mqtt.org/. |
| [3] | E. D. W. Dania Eridani, "Performance of Sensors Monitoring System using Raspberry Pi through MQTT Protocol," IEEE, 2018. |
| [4] | IBM, "MQTT publish/subscribe applications," 2022. |
| [5] | mosquitto.org, "mosquitto.org," mosquitto.org, [Online]. Available: https://mosquitto.org/man/mosquitto-conf-5.html. [Accessed 30 5 2022]. |
| [6] | D. C. M. C. Guilherme M. B. Oliveira and J. P. P. O. D. R. C. S. M. B. N. M. C. R. Ricardo J. B. V. M. Cavalcanti, "Comparison Between MQTT and WebSocket Protocols for IoT Applications Using ESP8266," IEEE, 2018. |
| [7] | Y. H. J. Z. D. F. Fu Chen, "A Review on the Study on MQTT Security Challenge," IEEE, 2020. |
| [8] | B. R. B. H. Syaiful Andy, "Attack scenarios and security analysis of MQTT communication protocol in IoT system," Researchgate, 2017. |
| [9] | B. Daniel, "Symmetric vs. Asymmetric Encryption: What's the Difference?," 4 5 2021. [Online]. Available: https://www.trentonsystems.com/blog/symmetric-vs-asymmetric-encryption. [Accessed 30 5 2022]. |
| [10] | X. B. C. T. R. A. H. H. A. S. Colin Urquhart, "Cyber-Security Internals of a Skoda Octavia vRS: A Hands on Approach," IEEE, 2019. |
| [11] | A. Mezher, "Enhanced RSA Cryptosystem based on Multiplicity of Public and Private Keys," Researchgate, 2018. |
| [12] | F. M. E. Valeriu Manuel Ionescu, "Investigating the performance of MicroPython and C on ESP32 and STM32 microcontrollers," IEEE, 2020. |